Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 329 (2004) 285–301

Theoretical Computer Science

www.elsevier.com/locate/tcs

A tight analysis and near-optimal instances of the algorithm of Anderson and Woll

Grzegorz Malewicz*,[1]

Department of Computer Science, University of Alabama, 116 Houser Hall, Tuscaloosa, AL 35487, USA

Received 2 July 2003; received in revised form 19 June 2004; accepted 6 October 2004
Communicated by P. Spirakis

Abstract

This paper shows an asymptotically tight analysis of the Certified Write-All algorithm called AWT that was introduced by Anderson and Woll, SIAM J. Comput. 26 (1997) 1277, and a method for creating near-optimal instances of the algorithm. This algorithm is the best known deterministic algorithm that can be used to simulate n synchronous parallel processors on n asynchronous processors. The algorithm is instantiated with q permutations on $\{1, \ldots, q\}$, where q can be chosen from a wide range of values. When implementing a simulation on a specific parallel system with n processors, one would like to select the best possible value of q and the best possible q permutations, in order to maximize the efficiency of the simulation.

This paper shows that work complexity of any instance of AWT is $\Theta\left(q^2/C \cdot n^{1+\log_q(C/q)}\right)$, where q is the number of permutations selected, and C is a value related to their combinatorial properties. The choice of q turns out to be critical for obtaining an instance of the AWT algorithm with near-optimal work. For any $\varepsilon > 0$, and any large enough n, work of any instance of the algorithm must be at least $n^{1+(1-\varepsilon)\sqrt{2 \ln \ln n/\ln n}}$. Under certain conditions, however, that q is about $e^{\sqrt{1/2 \ln n \ln \ln n}}$ and for infinitely many large enough n, this lower bound can be nearly attained by instances of the algorithm that use certain q permutations and have work at most $n^{1+(1+\varepsilon)\sqrt{2 \ln \ln n/\ln n}}$. The paper also shows

*Tel.: +1 205 348 4038; fax: +1 205 348 0219.
E-mail address: greg@cs.ua.edu (G. Malewicz).

[1]Extended abstract of this research appeared in the Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03).The work of Grzegorz Malewicz was done during a visit to the Supercomputing Technologies Group ("the Cilk Group"), Massachusetts Institute of Technology, headed by Prof. Charles E. Leiserson. Grzegorz Malewicz was visiting this group during the 2002/2003 academic year while in his final year of the Ph.D. program at the University of Connecticut, where his advisor was Prof. Alex Shvartsman. The work of the author was supported in part by the Singapore/MIT Alliance.

doi:10.1016/j.tcs.2004.10.015

a penalty for not selecting q well. When q is significantly away from $e^{\sqrt{1/2 \ln n \ln \ln n}}$, then work of any instance of the algorithm with this displaced q must be considerably higher than otherwise.

Keywords: Analysis of shared-memory algorithms; PRAM simulations; Certified Write-All

1. Introduction

This paper shows an asymptotically tight analysis of the Certified Write-All algorithm called AWT that was introduced by Anderson and Woll [1], and a method for creating near-optimal instances of the algorithm. In this algorithm n processors update n memory cells and then signal the completion of the updates. The algorithm is instantiated with q permutations, where q can be chosen from a wide range of values. This paper shows that the choice of q is critical for obtaining an instance of the AWT algorithm with near-optimal work.

Many existing parallel systems are asynchronous. However, writing correct parallel programs on an asynchronous shared memory system is often difficult, for example because of data races, which are difficult to detect in general [7,37]. When the instructions of a parallel program are written with the intention of being executed on a system that is synchronous, then it is easier for a programmer to write correct programs, because it is easier to reason about synchronous parallel programs than asynchronous ones. Therefore, in order to improve productivity in parallel computing, one could offer programmers the illusion that their programs run on a parallel system that is synchronous, while in fact the programs would be simulated on an asynchronous system.

Simulations of a parallel system that is synchronous on a system that is asynchronous have been studied for over a decade [3–6,10,14,16,19,21–24,32–34,40,41]. Simplifying considerably, such simulations assume that there is a system with p asynchronous processors, and the system is to simulate a program written for n synchronous processors. The simulations use three main ideas: idempotence, load balancing, and synchronization. Specifically, the execution of the program is divided into a sequence of phases. A phase executes an instruction of each of the n synchronous programs. The simulation executes a phase in two stages. First the n instructions are executed and the results are saved to a scratch memory. Only then cells of the scratch memory are copied back to desired cells of the main memory. This ensures that the result of the phase is the same even if multiple processors execute the same instruction in a phase, which may happen due to asynchrony. The p processors run a load balancing algorithm to ensure that the n instructions of the phase are executed quickly despite possibly varying speeds of the p processors. In addition, the p processors should be synchronized at every stage (twice per phase), so as to ensure that the simulated program proceeds in lock-step. Such simulation implements the PRAM model [15] on an asynchronous system.

One challenge in realizing the simulations is the development of efficient load-balancing and synchronization algorithms. This challenge is abstracted as the Certified Write-All (CWA) problem. In this problem, introduced in a slightly different form by Kanellakis and Shvartsman [19], there are p processors, an array w with n cells and a flag f, all initially 0,

and the processors must set the n cells of w to 1, and then set f to 1 (see [39] for a related problem of Collect). A simulation uses an algorithm that solves the CWA problem, and the overhead of the simulation depends on the efficiency of the algorithm. The efficiency of the algorithm is measured by *work* that is equal to the worst-case total number of instructions executed by the algorithm. In order to improve the efficiency of simulations, it is desirable to develop low-work algorithms that solve the CWA problem.

Deterministic algorithms that solve the CWA problem on an asynchronous system can be used to create simulations that have bounded worst-case overhead. Thus several deterministic algorithms have been studied [1,8,9,18,20,36]. The class of algorithms for the case when $p = n$ is especially interesting, because the algorithms have higher parallelism than algorithms for the case when $p \ll n$. When an algorithm for $n = p$ is used in a simulation, the simulation of a given synchronous program for $p = n$ processors may be faster, as compared to the simulation that uses an algorithm for $p \ll n$ processors, simply because in the former case more processors are available to simulate the program. However, the potential of producing a faster simulation can only be realized when the algorithm used has low work, so that not much computing resources are wasted during any simulation phase.

The best to date deterministic algorithm that solves the CWA problem on an asynchronous system for the case when $p = n$ was introduced by Anderson and Woll [1]. This algorithm is called AWT, and it generalizes the algorithm X of Buss et al. [8]. The AWT algorithm uses a list of q permutations on $\{1, \ldots, q\}$. Anderson and Woll introduced a notion of *contention* of a list of permutations, that is a value related to the number of left-to-right maxima in the permutations (see Section 2 for a formal definition). Anderson and Woll showed that for any $\varepsilon > 0$, there is a $q \in \mathbb{N}$, a list of q permutations with desired contention, and a constant c_q, such that the AWT algorithm for p processors and $n = p$ cells that uses the list, has work at most $c_q \cdot n^{1+\varepsilon}$. Note that this upper bound includes a multiplicative constant factor that is a function of q. The result that an $O(n^{1+\varepsilon})$ work algorithm can be found is very interesting from theoretical standpoint. However, a different search objective will occur when a simulation is developed for a specific parallel system.

A specific parallel system will have a fixed number p of processors. It is possible to create many instances of the AWT algorithm for these p processors and $n = p$ cells, that differ by the number q of permutations used to create an instance. It is possible that the work of these different instances is different. If this is indeed the case, then it is interesting to find the best possible value of q and the best possible q permutations, so as to create a relatively more efficient simulation on this parallel system.

Contributions: This paper shows an asymptotically tight analysis of the AWT algorithm of Anderson and Woll, and a method for creating near-optimal instances of the algorithm. In this algorithm, p processors update $n = p$ memory cells and then signal the completion of the updates. The algorithm is instantiated with q permutations on $\{1, \ldots, q\}$, where q can be chosen from a wide range of values. We show a bound of $\Theta\left(q^2/C \cdot n^{1+\log_q(C/q)}\right)$ on work of the AWT algorithm instantiated with a list of q permutations with contention C (appearing in Lemma 3.1). Then we demonstrate that the choice of q is critical for obtaining an instance of the AWT algorithm with near-optimal work. Specifically, we combine our bound with a lower bound on the contention of permutations given by Knuth [25] and Lovász [29], to show that for any $\varepsilon > 0$, the work of any instance must be at least $n^{1+(1-\varepsilon)\sqrt{2 \ln \ln n / \ln n}}$,

for any large enough n (appearing in Theorem 3.5). The resulting bound is nearly optimal: for any $\varepsilon > 0$ and for any m that is large enough, when $q = \lceil e^{\sqrt{1/2 \ln m \ln\ln m}} \rceil$, and $h = \lceil \sqrt{2 \ln m / \ln\ln m} \rceil$, then there exists an instance of the AWT algorithm for $p = q^h$ processors and $n = p$ cells that has work at most $n^{1+(1+\varepsilon)\sqrt{2 \ln\ln n / \ln n}}$ (appearing in Theorem 3.6). We also prove that there is a penalty if one selects a q that is too far away from $e^{\sqrt{1/2 \ln n \ln\ln n}}$. For any fixed $r \geqslant 2$, and any large enough n, work is at least $n^{1+r/3 \cdot \sqrt{2 \ln\ln n / \ln n}}$, whenever the AWT algorithm is instantiated with q permutations, such that $16 \leqslant q \leqslant e^{\sqrt{1/2 \ln n \ln\ln n}/(r \cdot \ln\ln n)}$ or $e^{r \cdot \sqrt{1/2 \ln n \ln\ln n}} \leqslant q \leqslant n$ (appearing in Preposition 3.7).

Paper organization: The remainder of the paper is organized as follows. In Section 2, we give definitions, report on some existing results on contention of permutations and present the AWT algorithm of Anderson and Woll. In Section 3, we show our asymptotically tight analysis of the AWT algorithm and an optimization argument. Finally, in Section 4, we conclude with future and related work.

2. Definitions and preliminaries

We consider an asynchronous PRAM. It consists of p processors and a shared memory. Any execution is modeled by a sequence of instructions issued by the processors. The executions are asynchronous, i.e. processors can work at arbitrarily varying paces. The shared memory cells are of the multi-reader multi-writer type, and reading and writing is instantaneous (see e.g., [2] for a formal definition, and also [5,11–13,17,28,32,38,42]). Any processor has a special *Halt* instruction that stops the operation of the processor.

We adopt the following definition of the Certified Write-All (CWA) problem: given an array $w[0, \ldots, n-1]$ with n cells and a flag f, all located in $n+1$ cells of shared memory and all initially 0, set the n cells of w to 1, and then set f to 1. An algorithm *solves* the CWA problem for p processors and n cells, if: (termination) each processor halts after having executed a finite number instructions, (certification) when any processor halts, the flag f has been set to 1, and (validity) when the flag f is set to 1, all n cells of w have been set to 1. The *work* complexity of a deterministic algorithm that solves the CWA problem for p processors and n cells is the maximum total number of instructions executed by the processors. (Work is a function of n and p.)

For a permutation ρ on $[q] = \{1, \ldots, q\}$, $\rho(v)$ is a *left-to-right maximum* [25] if it is larger than all of its predecessors; i.e. $\rho(v) > \rho(1), \rho(v) > \rho(2), \ldots, \rho(v) > \rho(v-1)$. The *contention* [1] of ρ with respect to a permutation α on $[q]$, denoted as $Cont(\rho, \alpha)$, is defined as the number of left-to-right maxima in the permutation $\alpha^{-1}\rho$ that is a composition of α^{-1} with ρ. For a list $R_q = \langle \rho_1, \ldots, \rho_q \rangle$ of q permutations on $[q]$ and a permutation α on $[q]$, the contention of R_q with respect to α is defined as $Cont(R_q, \alpha) = \sum_{v=1}^{q} Cont(\rho_v, \alpha)$. The contention of the list of permutations R_q is defined as $Cont(R_q) = \max_{\alpha \text{ on } [q]} Cont(R_q, \alpha)$.

Knuth [25] and Lovász [29] showed that the expectation of the number of left-to-right maxima in a random permutation on $[q]$ is H_q (H_q is the qth harmonic number). This immediately implies the following lower bound on contention of a list of q permutations on $[q]$.

```
AWT(R_q)
01    Traverse(h, λ)
02    set f to 1 and Halt

Traverse(i, s)
01    if i = 0 then
02        w[val(s)] := 1
03    else
04        j := q_i
05        for v := 1 to q
06            a := ρ_j(v)
07            if b_{s∘a} = 0 then
08                Traverse(i − 1, s ∘ a)
09                b_{s∘a} := 1
```

Fig. 1. The instance AWT(R_q) of an algorithm of Anderson and Woll, as executed by a processor with identifier $\langle q_1 \dots q_h \rangle$. The algorithm uses a list of q permutations $R_q = \langle \rho_1, \dots, \rho_q \rangle$.

Lemma 2.1 (*Knuth [25], Lovász [29]*). *For any list R_q of q permutations on $[q]$, $Cont(R_q) \geqslant q H_q > q \ln q$.*

Anderson and Woll [1] showed that for any q there is a list of q permutations with contention at most $3q H_q$. Since $H_q / \ln q$ tends to 1, as q tends to infinity, the following lemma holds.

Lemma 2.2 (*Anderson and Woll [1]*). *For any q that is large enough, there exists a list of q permutations on $[q]$ with contention at most $4 \cdot q \ln q$.*

We describe the algorithm AWT of Anderson and Woll [1] that solves the CWA problem when $p = n$. There are $p = q^h$ processors, $h \geqslant 1$, and the array w that has $n = p$ cells. The identifier of a processor is represented by a distinct string of length h over the alphabet $[q]$. The algorithm is instantiated with a list of q permutations $R_q = \langle \rho_1, \dots, \rho_q \rangle$ on $[q]$, and we write AWT(R_q) when we refer to the instance of algorithm AWT for a given list of permutations R_q. This list is available to every processor (in its local memory). Processors have access to a shared q-ary tree called *progress tree*. Each node of the tree is labeled with a string over alphabet $[q]$. Specifically, a string $s \in [q]^*$ that labels a node identifies the path from the root to the node (e.g., the root is labeled with the empty string λ, the leftmost child of the root is labeled with the string 1). For convenience, we say node s, when we mean the node labeled with a string s. Each node s of the tree, apart from the root, contains a *completion bit*, denoted by b_s, initially set to 0. Any leaf node s is canonically assigned a distinct number $val(s) \in \{0, \dots, n-1\}$.

The algorithm, shown in Fig. 1, starts by each processor calling procedure AWT(R_q). Each processor traverses the q-ary progress tree by calling a recursive procedure $Traverse(h, \lambda)$. When a processor visits a node s that is the root of a subtree of height i (the root of the progress tree has height h) the processor takes the ith letter j of its identifier (line `04`) and attempts to visit the children in the order established by the permutation ρ_j. The visit to a child $s \circ a$, for $a \in [q]$, *succeeds* only if the completion bit $b_{s \circ a}$ for this child is still 0 at the time of the attempt (line `07`). (The operation $\circ$ denotes concatenation, so $s \circ a$ is the concatenation of a string s with a letter a.) In such case, the processor recursively traverses the child subtree (line `08`), and later sets to one the completion bit of the child node (line `09`). When a processor visits a leaf s, the processor performs an assignment of 1 to the cell $val(s)$ of the array w. After a processor has finished the recursive traversal of the progress tree, the processor sets f to 1 and halts.

We give two technical lemmas that present closed forms of recursive equations. These equations arise in the analysis of the AWT algorithm carried out in the following section.

Lemma 2.3. *Let h and q be integers, $h \geqslant 1$, $q \geqslant 2$, and $k_1 + \cdots + k_q = c > 0$. Consider a recursive equation $W(0, r) = r$, and $W(i, r) = r \cdot q + \sum_{v=1}^{q} W(i-1, k_v \cdot r/q)$, when $i > 0$. Then for any r,*

$$W(h, r) = r\left(q \cdot \frac{(c/q)^h - 1}{c/q - 1} + (c/q)^h\right) .$$

Proof. First we observe that W is right-linear, i.e. that $W(i, z \cdot r) = z \cdot W(i, r)$, for any real *z*, which can be shown by induction on *i*. We use right-linearity to solve the recursive equation as follows:

$$\begin{aligned} W(h, r) &= r \cdot q + \sum_{v=1}^{q} k_v/q \cdot W(h-1, r) = r \cdot q + c/q \cdot W(h-1, r) \\ &= r \cdot q \frac{(c/q)^h - 1}{c/q - 1} + r(c/q)^h . \qquad \square \end{aligned}$$

Lemma 2.4. *Let h and q be integers, $h \geqslant 1$, $q \geqslant 2$, and for any string $s \in [q]^*$, $k_1^s + \cdots + k_q^s = c > 0$. Consider a recursive equation $V(s, r) = 3r$, for any string s of length h, and $V(s, r) = 7rq + \sum_{v=1}^{q} V(s \circ v, k_v^s \cdot r/q)$, for any string s of length less than h. Then for the empty string λ and any r*

$$V(\lambda, r) = r\left(7q \cdot \frac{(c/q)^h - 1}{c/q - 1} + 3(c/q)^h\right) .$$

Proof. First we observe that *V* is right-linear i.e., that $V(s, z \cdot r) = z \cdot V(s, r)$, which can be shown by a backward induction on the length of *s*, starting from length *h*. We also observe that the value of $V(s, r)$ is the same across all strings of the same length, i.e. for any *s* and s' of the same length, $V(s, r) = V(s', r)$. This can also be shown by backward induction, where the inductive step is

$$\begin{aligned} V(s, r) &= 7rq + \sum_{v=1}^{q} k_v^s/q \cdot V(s \circ v, r) = 7rq + \sum_{v=1}^{q} k_v^s/q \cdot V(s \circ 1, r) \\ &= 7rq + c/q \sum_{v=1}^{q} \cdot V(s' \circ 1, r) = V(s', r) . \end{aligned}$$

We use these two properties to solve the recursive equation as follows:

$$\begin{aligned} V(\lambda, r) &= 7r \cdot q + \sum_{v=1}^{q} k_v^{\lambda}/q \cdot V(v, r) = 7r \cdot q + c/q \cdot V(1, r) \\ &= 7r \cdot q \frac{(c/q)^h - 1}{c/q - 1} + 3r(c/q)^h . \qquad \square \end{aligned}$$

3. Tight analysis and near-optimal instances of AWT

This section presents an asymptotically tight analysis of the AWT algorithm of Anderson and Woll and a method for creating near-optimal instances of the algorithm. The main idea of this section is that for a fixed number p of processors and $n = p$ cells of the array w, work of any instance of the AWT algorithm depends on the number of permutations used by the instance, along with their contention, as shown by our analysis. This observation has several consequences. It turns out (not surprisingly) that work increases when contention increases, and conversely it becomes the lowest when contention is the lowest. Here a lower bound on contention of permutations given by Knuth [25] and Lovász [29] is very useful, because we can bound work of any instance from below, by an expression in which the value of contention of the list used in the instance is replaced with the value of the lower bound on contention. Then we study how the resulting lower bound on work depends on the number q of permutations on $[q]$ used by the instance. It turns out that there is a single value for q, where the bound attains the global minimum. Consequently, we obtain a lower bound on work that, for fixed n, is independent of both the number of permutations used and their contention. Our bound is near-optimal. We show that if we instantiate the AWT algorithm with about $e^{\sqrt{1/2 \ln n \ln \ln n}}$ permutations that have small enough contention, then work of the instance nearly matches the lower bound. Such permutations exist as shown by Anderson and Woll [1]. We also show that when we instantiate the AWT algorithm with much fewer or much more permutations, then work of the instance must be significantly greater than the work that can be achieved. Details of the overview follow.

We will present an asymptotically tight bound on work of any instance of the AWT algorithm. Our bound generalizes Theorem 5.2 of Anderson and Woll [1]. Our bound has an explicit constant which was hidden in the analysis given in Theorem 5.2. The constant will play a paramount role in the analysis presented in the remainder of the section.

Lemma 3.1. *Work W of the AWT algorithm for $p = q^h$ processors, $h \geqslant 1$, $q \geqslant 2$, and $n = p$ cells, instantiated with a list $R_q = \langle \rho_1, \ldots, \rho_q \rangle$ of q permutations on $[q]$, is bounded by*

$$\frac{c}{84} \cdot n^{1+\log_q \frac{Cont(R_q)}{q}} \leqslant W \leqslant c \cdot n^{1+\log_q \frac{Cont(R_q)}{q}},$$

where $c = \frac{28q^2}{Cont(R_q)}$.

Proof. The idea of the lemma is to carefully account for work spent on traversing the progress tree, and spent on writing to the array w. The lower bound will be shown by designing an execution during which the processors will traverse the progress tree in a specific, regular manner. This regularity will allow us to conveniently bound work inside a subtree from below by work done at the root of the subtree and work done by quite large number of processors that traverse the child subtrees in a regular manner. A similar recursive argument will be used to derive the upper bound.

Recall that in the asynchronous PRAM model an execution is a sequence of instructions. For convenience of the description, in the proof below, however, we will say that some

instructions are executed at the same instant. This extension could be modeled by allowing more than one instruction to be positioned at the same location in the sequence; an execution would then be a sequence of "piles" of instructions. A simple transformation allows us convert the argument given below for the extended model, to the asynchronous PRAM model. Indeed, the instructions stated to be executed at the same time will be either: reads from the same shared memory cell, or writes of the same value to the same cell, or operations on disjoint subsets of cells. Therefore, the instructions of a pile may be sequenced (or "flattened") in arbitrary order between the instructions of the previous and the next pile, to comply with the asynchronous PRAM model while preserving the outcome. Such transformation guarantees that the lemma holds in the model.

Consider any execution of the algorithm. We say that the execution is *regular at a node* s (recall that s is a string from $[q]^*$) iff the following three conditions hold:

(i) the r processors that ever visit the node during the execution, visit the node at the same time,
(ii) at that time, the completion bit of any node of the subtree of height i rooted at the node s is equal to 0,
(iii) if a processor visits the node s, and x is the suffix of length $h-i$ of the identifier of the processor, then the q^i processors that have x as a suffix of their identifiers, also visit the node during the execution.

We define $W(i,r)$ to be the largest number of instructions that r processors perform inside a subtree of height i, from the moment when they visit a node s that is the root of the subtree until the moment when each of the visitors finishes traversing the subtree, maximized across the executions that are regular at s and during which exactly r processors visit s (if there is no such execution, we put $-\infty$). Note that the value of $W(i,r)$ is well-defined, as it is independent of the choice of a subtree of height i (any pattern of traversals that maximizes the number of instructions performed inside a subtree, can be applied to any other subtree of the same height), and of the choice of the r visitors (suffixes of length $h-i$ do not affect traversal within the subtree). There exists an execution that is regular at the root of the progress tree, and so the value of $W(h,n)$ bounds work of $\mathrm{AWT}(R_q)$ from below.

We will show a recursive formula that bounds $W(i,r)$ from below. We do it by designing an execution recursively. The execution will be regular at every node of the progress tree. We start by letting the q^h processors visit the root at the same time. For the recursive step, assume that the execution is regular at a node s that is the root of a subtree of height i, and that exactly r processors visit the node. We first consider the case when s is an internal node i.e., when $i > 0$. Based on the ith letter of its identifier, each processor picks a permutation that gives the order in which completion bits of the child nodes will be read by the processor. Due to regularity, the r processors can be partitioned into q collections of equal cardinality, such that for any collection j, each processor in the collection checks the completion bits in the order given by ρ_j. Let for any collection, the processors in the collection check the bits of the children of the node in lock step (the collection behaves as a single "virtual" processor). Then, by Lemma 2.1 of Anderson and Woll [1], there is a pattern of delays so that every processor in some $k_v \geqslant 1$ collections succeeds in visiting the child $s \circ v$ of the node at the same time. Thus the execution is regular at any child node. The lemma also guarantees that $k_1 + \cdots + k_q = \mathit{Cont}(R_q)$, and that these $k_1, \ldots, k_q$

do not depend on the choice of the node s. Since each processor checks q completion bits of the q children of the node, the processor executes at least q instructions while traversing the node. Therefore, $W(i,r) \geqslant rq + \sum_{v=1}^{q} W(i-1, k_v \cdot r/q)$, for $i > 0$. Finally, suppose that s is a leaf, i.e. that $i = 0$. Then we let the r processors work in lock step, and so $W(0,r) \geqslant r$.

We can bound the value of $W(h,n)$ using Lemma 2.3, the fact that $h = \log_q n$, and that for any positive real a, $a^{\log_q n} = n^{\log_q a}$, as follows:

$$\begin{aligned} W(h,n) &\geqslant n \cdot \left(Cont(R_q)/q\right)^h \left(q \cdot \frac{1 - \left(q/Cont(R_q)\right)^h}{Cont(R_q)/q - 1} + 1 \right) \\ &= n^{1+\log_q (Cont(R_q)/q)} \left(q^2/Cont(R_q) \cdot \frac{1 - \left(q/Cont(R_q)\right)^h}{1 - q/Cont(R_q)} + 1 \right) \\ &> q^2/Cont(R_q) \cdot n^{1+\log_q (Cont(R_q)/q)} \left(1 - \left(q/Cont(R_q)\right)^h\right) \\ &\geqslant 1/3 \cdot q^2/Cont(R_q) \cdot n^{1+\log_q (Cont(R_q)/q)} , \end{aligned}$$

where the last inequality holds because for all $q \geqslant 2$, $q/Cont(R_q) \leqslant 2/3$, and $h \geqslant 1$.

The argument for proving an upper bound will be similar to the above argument for proving the lower bound. The main conceptual difference is that processors may write completion bits in different order for different internal nodes of the progress tree. Therefore, while the coefficients $k_1, \ldots, k_q$ were the same for each node during the analysis above, in the analysis of the upper bound presented now, each internal node s will have its own coefficients $k_1^s, \ldots, k_q^s$ that may be different for different nodes. To see this, take any execution and consider the root node $s = \lambda$. The r processors that visit the node satisfy condition (iii) and so they can be divided into q collections, each of cardinality r/q, where any processor in a collection j reads the completion bits in the order given by the permutation ρ_j. During the execution, the completion bits of the child nodes are set to 1 in some sequence, and let α_s be this sequence (α_s is a permutation on $[q]$). The argument of Anderson and Woll ensures that any processor from a collection j can only visit a child $s \circ v$, $1 \leqslant v \leqslant q$, when $\left(\alpha_s^{-1}\rho_j\right)(v)$ is a left-to-right maximum of the permutation $\alpha_s^{-1}\rho_j$. Let k_v^s be the number of permutations ρ_j such that $\left(\alpha_s^{-1}\rho_j\right)(v)$ is a left-to-right maximum. By the definition of contention, $k_1^s + \cdots + k_q^s \leqslant Cont(R_q)$. Work could only be increased if every processor from the collection j indeed visited every child node that corresponds to left-to-right maximum of $\alpha_s^{-1}\rho_j$ (a processor from the collection may not visit every child if another processor from the collection is quite fast and manages to set the completion bits to 1 before its peers check the bits). But if every processor from every collection visited every child node admissible by the sequence α_s, then the $k_v^s \cdot r/q$ processors that would visit the child node $s \circ v$ would satisfy condition (iii), and so could be divided into q collections of equal cardinality, and every processor from such collection would check completion bits according to a distinct permutation. Therefore, we could repeat the argument described for the root recursively for the child nodes. As a result, we obtain a recursive formula $V(s,r) \leqslant 3r$, when s is a string of length h, and $V(s,r) \leqslant 7rq + \sum_{v=1}^{q} V(s \circ v, k_v^s \cdot r/q)$, when s is a string of length less than h, while for all s, $k_1^s + \cdots + k_q^s \leqslant Cont(R_q)$. We can bound work of the given execution from above by $V(\lambda, n)$. The result now follows by applying Lemma 2.4 and observing that

$7q \cdot \frac{1-(q/Cont(R_q))^h}{1-q/Cont(R_q)} + 3$ can be bounded from above by $\frac{28q^2}{Cont(R_q)}$, because $q/Cont(R_q) \leqslant 2/3$ and $3 \leqslant 3q^2/Cont(R_q)$. □

How does the bound from the preceding lemma depend on contention of the list R_q? We should answer this question so that when we instantiate the AWT algorithm, we know whether to choose permutations with low contention or perhaps with high contention. The answer to the question may be not so clear at first, because for any given q, when we take a list R_q with lower contention, then although the exponent of n is lower, but the constant c is higher. In the lemma below we study this tradeoff, and demonstrate that it is indeed of advantage to choose lists of permutations with as small contention as possible.

Lemma 3.2. *The function* $c \mapsto q^2/c \cdot n^{\log_q c}$, *where* $c > 0$ *and* $n \geqslant q \geqslant 2$, *is a non-decreasing function of* c.

Proof. We consider a derivative

$$\frac{\partial}{\partial c}\left(q^2/c \cdot n^{\log_q c}\right) = -q^2/c^2 \cdot n^{\log_q c} + q^2/c \cdot (\ln n)/(c \ln q) \cdot n^{\log_q c}$$
$$= \left(-1 + \log_q n\right) q^2/c^2 \cdot n^{\log_q c}.$$

Recall that $n \geqslant q \geqslant 2$, and so $\log_q n \geqslant 1$. Thus the derivative is non-negative. □

This lemma, simple as it is, is actually quite useful. In several parts of the paper we use a list of permutations, for which we only know an upper bound or a lower bound on contention. The lemma allows us to bound work respectively from above or from below, even though we do not actually know the exact value of contention of the list.

We would like to find out how the lower bound on work depends on the choice of q. The subsequent argument shows that careful choice of the value of q is essential, in order to guarantee low work. We begin with two technical lemmas, the second of which bounds from below the value of a function occurring in Lemma 3.1.

The next lemma shows that an expression that is a function of x must vanish inside a "slim" interval.

Lemma 3.3. *Let* $\varepsilon > 0$ *be any fixed constant. Then for any large enough* n, *the expression* $x^2 - x + (1 - \ln x) \cdot \ln n$ *is negative when* $x = x_1 = \sqrt{1/2 \ln n \ln \ln n}$, *and positive when* $x = x_2 = \sqrt{(1+\varepsilon)/2 \ln n \ln \ln n}$.

Proof. The key idea of the proof is that x^2 creates in the expression a highest order summand with factor either $\frac{1}{2}$ or $(1+\varepsilon)/2$ depending on which of the two values of x we take, while $\ln x$ creates a summand of the same order with factor $\frac{1}{2}$ independent of the value of x. As a result, for the first value of x, the former "is less positive" than the latter "is negative", while when x has the other value, then the former "is more positive" than the latter "is negative". This intuition is made precise next.

We will show that the expression is negative when $x = \sqrt{\frac{1}{2} \ln n \ln \ln n}$. We split the expression into two parts: $x^2 - x$ and $(1 - \ln x) \ln n$, and compare their values. Obviously, $x^2 - x < \frac{1}{2} \ln n \ln \ln n$. We rewrite the second part as $(1 - \ln x) \ln n = -1/2 \ln n \ln \ln n$ $(1/2 \ln(1/2) - 1 + \frac{1}{2} \ln \ln \ln n) \ln n$, and observe that, for large enough n, the triple logarithm $\ln \ln \ln n$ is large enough so that any negative constant summand in the parenthesis becomes balanced by a positive summand, and so the expression inside parenthesis becomes positive. As a result, for large enough n, the second part is smaller or equal to $-\frac{1}{2} \ln n \ln \ln n$. This means that the sum of the first part and the second part is negative, as desired.

We will show that the expression is positive when $x = \sqrt{(1+\varepsilon)/2 \ln n \ln \ln n}$. Since the summand x of the first part is of lower order than the summand x^2, we know that for large enough n, $x^2 - x > (1 + \varepsilon/2)/2 \ln n \ln \ln n$ (recall that $\varepsilon > 0$ is a fixed constant). Now a key observation is that the multiplier of the highest order summand $\ln n \ln \ln n$ of the second part is equal to $\frac{1}{2}$, and not $(1 + \varepsilon)/2$, because it is obtained by taking a logarithm of a square root. Thus the second part $(1 - \ln x) \ln n = -\frac{1}{2} \ln n \ln \ln n - \left(\frac{1}{2} \ln((1 + \varepsilon)/2) - 1 + \frac{1}{2} \ln \ln \ln n\right) \ln n$ is more than $-(1+\varepsilon/2)/2 \ln n \ln \ln n$, for large enough n. This completes the proof. □

Lemma 3.4. *Let $\varepsilon > 0$ be any fixed constant. Then for any large enough n, the value of the function $f : [\ln 3, \ln n] \to \mathbb{R}$, defined as $f(x) = e^x/x \cdot n^{\ln x/x}$, is bounded from below by $f(x) \geqslant n^{(1-\varepsilon)\sqrt{2 \cdot \ln \ln n / \ln n}}$.*

Proof. We shall show the lemma by reasoning about the derivative of f. We will see that it contains two parts: one that is strictly convex, and the other that is strictly concave. This will allow us to conveniently reason about the sign of the derivative, and where the derivative vanishes. As a result, we will ensure that there is only one local minimum of f in the interior of the domain. An additional argument will ascertain that the values of f at the boundary are larger than the minimum value attained in the interior.

Let us investigate where the derivative

$$\frac{\partial f}{\partial x} = e^x n^{\ln x/x}/x^3 \cdot \left(x^2 - x + (1 - \ln x) \ln n\right)$$

vanishes. It happens only for such x, for which the parabola $x \mapsto x^2 - x$ "overlaps" the logarithmic plot $x \mapsto \ln n \ln x - \ln n$. We notice that the parabola is strictly convex, while the logarithmic plot is strictly concave. Therefore, we conclude that one of the three cases must happen: plots do not overlap, plots overlap at a single point, or plots overlap at exactly two distinct points. We shall see that the latter must occur for any large enough n.

We will see that the plots overlap at exactly two points. Note that when $x = \ln 3$, then the value of the logarithmic plot is negative, while the value of the parabola is positive. Hence the parabola is "above" the logarithmic plot at the point $x = \ln 3$ of the domain. Similarly, it is "above" the logarithmic plot at the point $x = \ln n$, because for this x the highest order summand for the parabola is $\ln^2 n$, while it is only $\ln n \ln \ln n$ for the logarithmic plot. Finally, we observe that when $x = \sqrt{\ln n}$, then the plots are "swapped": the logarithmic

plot is "above" the parabola, because for this x the highest order summand for the parabola is $\ln n$, while the highest order summand for the logarithmic plot is as much as $\frac{1}{2} \ln n \ln \ln n$. Therefore, for any large enough n, the plots must cross at exactly two points in the interior of the domain.

Now we are ready to evaluate the monotonicity of f. By inspecting the sign of the derivative, we conclude that f increases from $x = \ln 3$ until the first point, then it decreases until the second point, and then it increases again until $x = \ln n$. This holds for any large enough n. This pattern of monotonicity simplifies case analysis when establishing a lower bound on f.

There is only one local minimum of f in the interior of the domain. The function f attains a local minimum at the second point, and Lemma 3.3 teaches us that this point is in the range between $x_1 = \sqrt{\frac{1}{2} \ln n \ln \ln n}$ and $x_2 = \sqrt{(1+\varepsilon)/2 \ln n \ln \ln n}$. For large enough n, we can bound the value of the local minimum from below by $f_1 = e^{x_1}/x_2 \cdot n^{\ln x_1/x_2}$. We can further weaken this bound as

$$f_1 = n^{-\ln x_2/\ln n + \ln x_1/x_2 + x_1/\ln n} \geqslant n^{-\ln x_2/\ln n + 1/2 \ln \ln n/x_2 + \sqrt{1/2 \ln \ln n/\ln n}} \geqslant n^{(1-\varepsilon)\sqrt{2 \cdot \ln \ln n/\ln n}},$$

where the first inequality holds because for large enough n, $\ln(\frac{1}{2} \ln \ln n)$ is positive, while the second inequality holds because for large enough n, $\ln x_2 \leqslant \ln \ln n$, and $1/\sqrt{1+\varepsilon} \geqslant 1-\varepsilon$, and for large enough n, $\sqrt{\frac{1}{2} \ln \ln n/\ln n} - \ln \ln n/\ln n$ is larger than $\sqrt{1/(2+2\varepsilon) \ln \ln n/\ln n}$.

Finally, we note that the values attained by f at the boundary are strictly larger than the value attained at the second point. Indeed, $f(\ln n)$ is strictly greater, because the function strictly increases from the second point towards $\ln n$. In addition, $f(\ln 3)$ is strictly greater because it is at least $n^{1.08}$, while the value attained at the second point is bounded from above by n raised to a power that tends to 0 as n tends to ∞ (in fact it suffices to see that the exponent of n in the bound on f_1 above, tends to 0 as n tends to ∞).

This completes the argument showing a lower bound on f. □

The following two theorems show that we can construct an instance of AWT that has the exponent for n arbitrarily close to the exponent that is required, provided that we choose the value of q carefully enough.

Theorem 3.5. *Let $\varepsilon > 0$ be any fixed constant. Then for any n that is large enough, any instance of the AWT algorithm for $p = n$ processors and n cells has work at least $n^{1+(1-\varepsilon)\sqrt{2 \ln \ln n/\ln n}}$.*

Proof. This theorem is proven by combining the results shown in the preceding lemmas. Take any AWT algorithm for n cells and $p = n$ processors instantiated with a list R_q of q permutations on $[q]$. By Lemma 3.1, work of the instance is bounded from below by the expression $q^2/(3Cont(R_q)) \cdot n^{1+\log_q(Cont(R_q)/q)}$. By Lemma 3.2, we know that this expression does not increase when we replace $Cont(R_q)$ with a number that is smaller

or equal to $Cont(R_q)$. Indeed, this is what we will do. By Lemma 2.1, we know that the value of $Cont(R_q)$ is bounded from below by $q \ln q$. Hence work of the AWT is at least $n/3 \cdot q/\ln q \cdot n^{\ln \ln q/\ln q}$.

Now we would like to have a bound on this expression that does not depend on q. This bound should be fairly tight so that we can later find an instance of the AWT algorithm that has work close to the bound. Let us make a substitution $q = e^x$. We can use Lemma 3.4 with $\varepsilon/2$ to bound the expression from below as desired, for large enough n, when q is in the range from 3 to n. What remains to be checked is how large work must be when the AWT algorithm is instantiated with just two permutations (i.e., when $q = 2$). In this case we know that contention of any list of two permutations is at least 3, and so work is bounded from below by n raised to a fixed power strictly greater than 1. Thus the lower bound holds for large enough n. □

Theorem 3.6. *Let $\varepsilon > 0$ be any fixed constant. Then for any large enough m, when $q = \lceil e^{\sqrt{1/2 \ln m \ln \ln m}} \rceil$, and $h = \lceil \sqrt{2 \ln m/\ln \ln m} \rceil$, there exists an instance of the AWT algorithm for $p = n = q^h$ processors and n cells that has work at most $n^{1+(1+\varepsilon)\sqrt{2 \ln \ln n/\ln n}}$.*

Proof. Had it not been for the ceilings in the definitions of q and h, the result would have been immediate. The main problem that we are facing is that taking ceiling could make q too far away from the best possible choice for q, and so work of the resulting algorithm could be too large compared to the lower bound on work of Theorem 3.5. However, this cannot happen, as we will see shortly. Intuitively, this is because h and q are quite small, and so q^h is close to the q^h with ceilings dropped. This intuition is formally shown next.

We construct a specific instance of the AWT algorithm and bound its work from above. By Lemma 2.2, for any q that is large enough, there exists a list of q permutations on $[q]$ with contention at most $4q \ln q$. Thus, by Lemmas 3.1 and 3.2, work W of the AWT algorithm instantiated with this list is at most $W \leqslant 28q/(4 \ln q) \cdot n^{1+\ln(4 \ln q)/\ln q}$, for any m that is large enough. We now apply a series of algebraic manipulations to bound this expression from above, for large enough m. Specifically, W can be bounded as

$$\begin{aligned} W &\leqslant 28 \cdot n^{1+\ln(4 \ln q)/\ln q + \ln q/\ln n} \\ &\leqslant n^{1+\ln(8\sqrt{1/2 \ln m \ln \ln m})/\sqrt{1/2 \ln m \ln \ln m} + (1+\ln 28+\sqrt{1/2 \ln m \ln \ln m})/\ln m} \\ &\leqslant n^{1+\left(1+\sqrt{\ln \ln \ln m/\ln \ln m}\right)\sqrt{2 \ln \ln m/\ln m}}, \end{aligned}$$

where the second inequality holds because for large enough m, $q \leqslant e^{1+\sqrt{1/2 \ln m \ln \ln m}}$ is at most $e^{2 \cdot \sqrt{1/2 \ln m \ln \ln m}}$, and because $m \leqslant n$, while the third inequality holds because for large enough m, $\ln(8\sqrt{1/2 \ln \ln m})/\sqrt{1/2 \ln m \ln \ln m} + (1 + \ln 28)/\ln m$ is at most $\ln \ln \ln m/\sqrt{1/2 \ln m \ln \ln m}$.

Our goal now is to replace every occurrence of m above with n. Since $m \leqslant n$, we can easily do the substitution in the enumerator. However, we also have an expression $1/\ln m$, where such substitution could decrease the exponent. In order to alleviate this problem, we will show that n and m are close to each other. Let $\bar{q}$ and $\bar{h}$ be equal to q and h, respectively, except for the ceilings dropped, i.e. $\bar{q} = e^{\sqrt{1/2 \ln m \ln \ln m}}$, and $\bar{h} =$

$\sqrt{2 \ln m / \ln \ln m}$. We can trivially bound n from above by m^3, because $\bar{q}^2 \geqslant q$, for large enough m. However, any upper bound of m raised to a power bounded away from 1 is not satisfactory, as it will boost the constant 1 in the $(1 + \sqrt{\ln \ln \ln m}/\ln \ln m)$ factor in the expression above. Therefore, we need a tighter upper bound, and we will develop it now. We first note that $((\bar{q}+1)/\bar{q})^{\bar{h}} \leqslant e^{\bar{h}/\bar{q}} \leqslant m^{\bar{h}/\ln m} \leqslant m^{\sqrt{2/(\ln m \ln \ln m)}}$, and that for large enough m, $\bar{q} + 1 \leqslant \bar{q}^2 = m^{2 \ln \bar{q}/\ln m} \leqslant m^{\sqrt{2 \ln \ln m / \ln m}}$. Using the fact that, for large enough m, $\ln(m^3) \ln \ln(m^3) \leqslant 4 \ln m \ln \ln m$, and $n \leqslant m^3$, we obtain two bounds: $1/(\ln m \ln \ln m) \leqslant 4/(\ln n \ln \ln n)$ and $1/\ln m \leqslant 3/\ln n$. These two bounds can be applied to replace m with n in the former two bounds, and we experience a slight weakening of the former bounds: $((\bar{q}+1)/\bar{q})^{\bar{h}} \leqslant m^{\sqrt{8/(\ln n \ln \ln n)}}$, and $\bar{q} + 1 \leqslant m^{\sqrt{6 \ln \ln n / \ln n}}$. We combine the latest two bounds to show that n is bounded from above by m raised to a power that tends to 1, as n tends to infinity. Specifically, $n \leqslant (\bar{q}+1)^{\bar{h}+1} = \bar{q}^{\bar{h}}((\bar{q}+1)/\bar{q})^{\bar{h}} \cdot (\bar{q}+1) \leqslant m^{1+8\sqrt{\ln \ln n/\ln n}}$. This ensures that $1/\ln m \leqslant (1 + 8\sqrt{\ln \ln n/\ln n})/\ln n$, for any large enough m. This bound allows us to replace m with n in the expression above, while maintaining the $(1 + o(1))$ multiplicative factor. Thus the result follows. □

The preceding two theorems teach us that when q is selected carefully, we can create an instance of the AWT algorithm that is nearly optimal. A natural question that one immediately asks is: what if q is *not* selected well enough? Lemmas 3.1 and 3.2 teach us that lower bound on work of an instance of the AWT algorithm depends on the number q of permutations on $[q]$ used by the instance. On one extreme, if q is a constant that is at least 2, then work must be at least n to some exponent that is greater than 1 and that is bounded away from 1. On the other extreme, if $q = n$, then work must be at least n^2. In the "middle", when q is about $e^{\sqrt{1/2 \ln n \ln \ln n}}$, then the lower bound is the weakest, and we can almost attain it as shown in the preceding two theorems. Suppose that we chose the value of q slightly away from the value $e^{\sqrt{1/2 \ln n \ln \ln n}}$. By how much must work be increased as compared to the lowest possible value of work? Although one can carry out a more precise analysis of the growth of a lower bound as a function of q, we will be contented with the following result, which already establishes a gap between the work possible to attain when q is chosen well, and the work required when q is not chosen well.

Proposition 3.7. *Let $r \geqslant 2$ be any fixed constant. For any large enough n, if the AWT algorithm is instantiated with q permutations on $[q]$, such that $16 \leqslant q \leqslant e^{\sqrt{1/2 \ln n \ln \ln n}/(r \cdot \ln \ln n)}$ or $e^{r \cdot \sqrt{1/2 \ln n \ln \ln n}} \leqslant q \leqslant n$, then its work is at least $n^{1+r/3 \cdot \sqrt{2 \ln \ln n/\ln n}}$.*

Proof. By Lemmas 3.1, 3.2, and 2.1, work of the AWT algorithm instantiated with any list of q permutations on $[q]$ is at least

$$q/\ln q \cdot n^{1+\ln \ln q/\ln q - \ln 3/\ln n}.$$

Suppose that q falls into the first interval. By taking logarithm of both sides of the inequality $q \leqslant e^{\sqrt{1/2 \ln n \ln \ln n}/(r \cdot \ln \ln n)}$, we obtain that $r \cdot \sqrt{2 \ln \ln n/\ln n}$ is at most $1/\ln q$. Recall that $q \geqslant 16$, and so $\ln \ln q \geqslant 1$. Therefore, we can multiply the right-hand side of the former inequality by $\ln \ln q$ without violating the inequality, and obtain $r/2 \cdot$

$\sqrt{2 \ln \ln n / \ln n} \leqslant \ln \ln q / \ln q$, as desired. Now suppose that q falls into the second interval. We obtain the desirable bound by observing that then $q \geqslant n^{r \cdot \sqrt{1/2 \ln n \ln \ln n}/\ln n} = n^{r/2 \cdot \sqrt{2 \ln \ln n / \ln n}}$. □

4. Conclusions, future work and related work

We have seen an asymptotically tight analysis of the AWT algorithm, and that the choice of the number of permutation is critical for obtaining an instance of the algorithm with near-optimal work. Specifically, when the algorithm is instantiated with about $e^{\sqrt{1/2 \ln n \ln \ln n}}$ permutations, then work of an instance can be near-optimal, while when q is significantly away from $e^{\sqrt{1/2 \ln n \ln \ln n}}$, then work of any instance of the algorithm with this displaced q must be considerably higher than otherwise.

The main open problem is to determine an asymptotically optimal bound on work complexity of deterministic algorithms for the Certified Write-All. There is a known lower bound of $\Omega(n \log n)$ on work of $p = n$ processors on n cells. This bound is attained by a randomized algorithm. Is there higher lower bound for deterministic algorithms? How far is the AWT algorithm from being optimal? The author of this paper showed a work-optimal deterministic algorithm for a nontrivial number of $p < n^{1/5}$ processors. This result appeared in the doctoral dissertation of the author [31] and also as [30]. Lately, the paper of Kowalski and Shvartsman [27] improved this result by showing an optimal algorithm for $p < n^{1/(2+\varepsilon)}$ processors; the algorithm uses q permutations with contention $O(q \log q)$.

It is an open problem how to efficiently and deterministically construct q permutations with contention $O(q \cdot \mathit{polylog}(q))$. The paper of Chlebus et al. [9] discusses analytical and experimental results indicating that such construction may be possible. In a recent paper, Kowalski and Shvartsman [26] extended the notion of contention of a list of permutations to d-contention that measures the worst case total number of elements that have fewer than d larger predecessors (and so the contention of Anderson and Woll is equal to the 1-contention of Kowalski and Shvartsman). They showed that d-contention plays an important role in designing message-passing asynchronous algorithms for performing independent tasks, such that the algorithms have low work in the presence of message delays.

There are several follow-up research directions on how one could try to generalize and fine-tune the AWT algorithm. Any AWT algorithm has a progress tree with internal nodes of fanout q. One could consider generalized AWT algorithms where fanout does not need to be uniform. Suppose that a processor that visits a node of height i, uses a collection $R^i_{q(i)}$ of $q(i)$ permutations on $[q(i)]$. Now we could choose different values of $q(i)$ for different heights i. Does this technique enable any reduction of work as compared to the case when $q = q(1) = \cdots = q(h)$? What are the best values for $q(1), \ldots, q(h)$ as a function of n? Suppose that we are given a relative cost κ of performing a write to the cell of the array w, compared to the cost of executing any other instruction. What is the shape of the progress tree that minimizes work? These questions give rise to more complex optimization problems, which would be interesting to solve.

Acknowledgements

The author thanks Charles Leiserson for an invitation to join the Supercomputing Technologies Group, and Dariusz Kowalski, Arnold Rosenberg, Larry Rudolph, and Alex Shvartsman for their comments that improved the quality of the presentation. The author acknowledges a Theoretical Computer Science reviewer who provided very helpful and detailed suggestions on how to improve the paper.

References

[1] R.J. Anderson, H. Woll, Algorithms for the certified write-all problem, SIAM J. Comput. 26 (5) (1997) 1277–1283.
[2] J. Aspnes, M. Herlihy, Wait-free data structures in the asynchronous PRAM model, in: 2nd Symp. on Parallel Algorithms and Architectures SPAA'90, 1990, pp. 340–349.
[3] Y. Aumann, M.A. Bender, L. Zhang, Efficient execution of nondeterministic parallel programs on asynchronous systems, Inform. and Comput. 139 (1) (1997) 1–16.
[4] Y. Aumann, Z.M. Kedem, K.V. Palem, M.O. Rabin, Highly efficient asynchronous execution of large-grained parallel programs, in: 34th IEEE Symp. on Foundations of Computer Science FOCS'93, 1993, pp. 271–280.
[5] Y. Aumann, M.O. Rabin, Clock construction in fully asynchronous parallel systems and PRAM simulation, Theoret. Comput. Sci. 128 (1994) 3–30.
[6] A. Baratloo, P. Dasgupta, Z.M. Kedem, CALYPSO: a novel software system for fault-tolerant parallel processing on distributed platforms, in: 4th Internat. Symp. on High Performance Distributed Computing HPDC'95, 1995, pp. 122–129.
[7] A.J. Bernstein, Program analysis for parallel processing, IEEE Trans. Electronic Comput. EC-15 (5) (1966) 757–762.
[8] J. Buss, P.C. Kanellakis, P.L. Ragde, A.A. Shvartsman, Parallel algorithms with processor failures and delays, J. Algorithms 20 (1996) 45–86.
[9] B. Chlebus, S. Dobrev, D. Kowalski, G. Malewicz, A. Shvartsman, I. Vrto, Towards practical deterministic write-all algorithms, in: 13th Symp. on Parallel Algorithms and Architectures SPAA'01, 2001, pp. 271–280.
[10] B.S. Chlebus, A. Gambin, P. Indyk, PRAM computations resilient to memory faults, in: 2nd European Symp. on Algorithms ESA'94, 1994, pp. 401–412.
[11] R. Cole, O. Zajicek, The APRAM: incorporating asynchrony into the PRAM model, in: 2nd ACM Symp. on Parallel Algorithms and Architectures SPAA'89, 1989, pp. 169–178.
[12] R. Cole, O. Zajicek, The expected advantage of asynchrony, in: 3rd ACM Symp. on Parallel Algorithms and Architectures SPAA'90, 1990, pp. 85–94.
[13] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. van Eicken, LogP: towards a realistic model of parallel computation, in: 4th ACM Principles and Practices of Parallel Programming, 1993, pp. 1–12.
[14] P. Dasgupta, Z.M. Kedem, M.O. Rabin, Parallel processing on networks of workstations: a fault-tolerant, high performance approach, in: 15th Internat. Conf. on Distributed Computing Systems ICDCS'95, 1995, pp. 467–474.
[15] S. Fortune, J. Wyllie, Parallelism in random access machines, in: 10th ACM Symp. on Theory of Computing, 1978, pp. 114–118.
[16] J.D. Garofalakis, P.G. Spirakis, B. Tampakas, S. Rajsbaum, Tentative and definite distributed computations: an optimistic approach to network synchronization, Theoret. Comput. Sci. 128 (1&2) (1994) 63–74.
[17] P.B. Gibbons, A more practical PRAM model, in: 2nd ACM Symp. on Parallel Algorithms and Architectures SPAA'89, 1989, pp. 158–168.
[18] J.F. Groote, W.H. Hesselink, S. Mauw, R. Vermeulen, An algorithm for the asynchronous write-all problem based on process collision, Distributed Comput. 14 (2) (2001) 75–81.
[19] P.C. Kanellakis, A.A. Shvartsman, Efficient parallel algorithms can be made robust, Distributed Comput. 5 (4) (1992) 201–217.

[20] P.C. Kanellakis, A.A. Shvartsman, Fault-Tolerant Parallel Computation, Kluwer Academic Publishers, Dordrecht, 1997.
[21] R.M. Karp, M. Luby, F. Meyer auf der Heide, Efficient PRAM simulation on a distributed memory machine, Algorithmica 16 (1996) 517–542.
[22] Z.M. Kedem, K.V. Palem, M.O. Rabin, A. Raghunathan, Efficient program transformations for resilient parallel computation via randomization (preliminary version), in: 24th ACM Symp. on Theory of Computing STOC'92, 1992, pp. 306–317.
[23] Z.M. Kedem, K.V. Palem, A. Raghunathan, P.G. Spirakis, Combining tentative and definite executions for very fast dependable parallel computing (extended abstract), in: 23rd ACM Symp. on Theory of Computing STOC'91, 1991, pp. 381–390.
[24] Z.M. Kedem, K.V. Palem, P.G. Spirakis, Efficient robust parallel computations, in: 22nd ACM Symp. on Theory of Computing STOC'90, 1990, pp. 138–148.
[25] D.E. Knuth, The Art of Computer Programming, Vol. 3, third ed., Addison-Wesley, Reading, MA, 1998.
[26] D.R. Kowalski, A.A. Shvartsman, Performing work with asynchronous processors: message-delay-sensitive bounds, in: 22nd ACM Symp. on Principles of Distributed Computing PODC'03, 2003, pp. 265–274.
[27] D.R. Kowalski, A.A. Shvartsman, Writing-all deterministically and optimally using a non-trivial number of asynchronous processors, 16th ACM Symp. on Parallelism in Algorithms and Architectures SPAA'04, 2004, to appear.
[28] C.P. Kruskal, L. Rudolph, M. Snir, A complexity theory of efficient parallel algorithms, Theoret. Comput. Sci. 71 (1) (1990) 95–132.
[29] L. Lovász, Combinatorial Problems and Exercises, 2nd ed., North-Holland, Amsterdam, 1993.
[30] G. Malewicz, A work-optimal deterministic algorithm for the certified write-all problem with a nontrivial number of asynchronous processors, SIAM J. Comput., to appear.
[31] G. Malewicz, Distributed scheduling for disconnected cooperation, Doctoral Dissertation, University of Connecticut, 2003.
[32] C. Martel, A. Park, R. Subramonian, Work-optimal asynchronous algorithms for shared memory parallel computers, SIAM J. Comput. 21 (6) (1992) 1070–1099.
[33] C. Martel, R. Subramonian, How to emulate synchrony, Technical Report CSE-90-26, UC Davis, 1990.
[34] C. Martel, R. Subramonian, Asynchronous PRAM algorithms for list ranking and transitive closure, Internat. Conf. on Parallel Processing ICPP'90, Vol. 3, 1990, pp. 60–63.
[35] C. Martel, R. Subramonian, On the complexity of certified write-all algorithms, J. Algorithms 16 (3) (1994) 361–387.
[36] J. Naor, R.M. Roth, Constructions of permutation arrays for certain scheduling cost measures, Random Struct. Algorithms 6 (1) (1995) 39–50.
[37] R.H.B. Netzer, B.P. Miller, On the complexity of event ordering for shared-memory parallel program executions, Internat. Conf. on Parallel Processing ICPP'90, Vol. 2, 1990, pp. 93–97.
[38] N. Nishimura, Asynchronous shared memory parallel computation, in: 3rd ACM Symp. on Parallel Algorithms and Architectures SPAA'90, 1990, pp. 76–84.
[39] M.E. Saks, N. Shavit, H. Woll, Optimal time randomized consensus—making resilient algorithms fast in practice, in: 2nd ACM-SIAM Annu. Symp. on Discrete Algorithms SODA'91, 1991, pp. 351–362.
[40] A.A. Shvartsman, Achieving optimal CRCW PRAM fault-tolerance, Inform. Process. Lett. 39 (2) (1991) 59–66.
[41] R. Subramonian, Designing synchronous algorithms for asynchronous processors, in: 4th ACM Symp. on Parallel Algorithms and Architectures SPAA'92, 1992, pp. 189–198.
[42] L. Valiant, A bridging model for parallel computation, Comm. ACM 33 (8) (1990) 103–111.